SOURCE CODE APPENDIX

The algorithm is implemented within the runAsServer function of the DEM class and some other functions called within this function. These functions are provided below.

The software code is presented in the Courier font. Code corresponding to the specific lines of the algorithm (see the section heading "Algorithm for Dynamic Slip Control") are presented in the **Bold Courier** font. Commentary is presented in the *Italic Roman* font.

The DEM::runAsServer Function

```
void
DEM::runAsServer()
{
  Alert *alert;
  int selection;
  int i;

#ifndef EMBEDDED
  fd_set rfd;
  char error[256];

  DEM::setPrompt();
  DEM::makeDataDirectories();
#endif
```

The following high-lighted code fragment corresponds to line 1 of the algorithm (i.e, $x = x_0$).

```
  // Performs user-specified application initializations
  appInit();

  // Insantiates and initializes user-defined components
for server
  // initial state
#ifndef EMBEDDED
  if (callAppInitComponents)
#endif
    appInitComponents();

#ifndef EMBEDDED

  if (loadFile)
    DEM::loadComponents(loadFile);
```

```
#endif // EMBEDDED
```

The following high-lighted code fragment corresponds to line 2 of the algorithm [i.e, t = current_time() *].*

```
    // Time variables are initialized with system time
    systemStartTime = currentApplicationTime =
transitionRealTime = dem_time();
```

The following high-lighted code fragment corresponds to line 3 of the algorithm (i.e., k = 1*).*

```
    transitionId = 1;
```

The following high-lighted code fragment corresponds to line 4 of the algorithm (i.e., forever { *).*

```
    while(1) {
```

The following high-lighted code fragment corresponds to line 5 of the algorithm [i.e., Δ = next_event_time(k, x) *]. The function DEM::findProaction is described in a following section.*

```
        // Finds the earliest proaction (if any) and stores
    it in
        // DEM::scheduledTransition. It also assigns the
    right values to
        // DEM::firingDelay and DEM::scheduledComponent
        DEM::findProaction();
```

The following high-lighted code fragment corresponds to line 6 of the algorithm [i.e., τ = set_interrupt_timer(Δ - (current_time() - $\tilde{t}$)) *]. The DEM::waitForInterrupt functions are described in a following section.*

```
#ifdef WIN

#ifndef EMBEDDED
    selection = DEM::waitForInterrupt(&rfd);
#else // EMBEDDED
    selection = DEM::waitForInterrupt();
#endif
```

```
#else // WIN

#ifndef EMBEDDED
      if (enterprise())
        selection = DEM::waitForInterrupt(&rfd);
      else
#endif // EMBEDDED
        selection = DEM::waitForInterrupt();

#endif // WIN
```

The following high-lighted code fragment corresponds to lines 7 and 9 of the algorithm [i.e., $\tilde{t}_{next}$ = current_time() and $\tilde{t} = \tilde{t}_{next}$ *]. The DEM::updateTimeVariables function is described in a later section.*

```
    // Updates time variables according to rt and sim
options.
    DEM::updateTimeVariables (selection);
```

The following high-lighted code fragment corresponds to line 8 of the algorithm [i.e., x = update(k, ($\tilde{t}_{next}$ - $\tilde{t}$)) *]. The update function is described in a later section.*

```
    // If time has elapsed components are updated
    if (firingDelay > 0)
      for(i = 0; i < realtimeComponents.size; i++)
        realtimeComponents.elements[i]-
>update(firingDelay);
```

The following high-lighted code fragment corresponds to line 10 of the algorithm

[i.e., x = compute(τ, k, x) *].*

```
    // The action associated with the scheduled
transition is executed and
    // the return event is stored in event.
    Event *event = (scheduledComponent-
>*scheduledComponent->
                a()[scheduledTransition])(alert);
```

The following high-lighted code fragment corresponds to line 11 of the algorithm

(i.e., $k = k + 1$ *).*

```
    transitionId += 1;
```

The following high-lighted code fragment corresponds to line 12 of the algorithm (i.e., }).

```
  }
}
```

The DEM::findProaction Function

```
 void
 DEM::findProaction()
 {
 #ifndef EMBEDDED
   char error[256];
 #endif

   firingDelay = -1;
   scheduledComponent = 0;
   scheduledTransition = -1;

   /* For the first component in a transient state */
   while(transientStateComponents.size > 0) {
     Component *c = transientStateComponents.elements[0];

     if (c->transientStates()[c->q] == 0) {
       transientStateComponents.remove(c);
       continue;
     }

     int n = c->npro(), *p = c->pro();
     double (Component::**g)() = c->g();
     Transition *t = c->t();
     double d;

     /* ... for all proactions in current component starting in
state q */
     for(int j=0; j<n; j++) {
       if (c->q != t[p[j]].from)
      continue;
       d = (c->*g[p[j]])();
       if (d == 0) {
      scheduledComponent = c;
      scheduledTransition = p[j];
      firingDelay = d;
      transientStateComponents.remove(c);
      return;
       }
     }
```

```
      /* current component is in transient state but
         no exiting proaction is enabled */
  #ifndef EMBEDDED
      sprintf(error, "DEM-findProaction: Component %d/%ld is in
  transient state %d but no outgoing proaction is enabled.\n",
  c->cid(), c->id, c->q);
      writeErrorLog(error);
  #endif
      transientStateComponents.remove(c);
      // Force c to go into its error state
      c->q = 0;
    }

   /* For all proactive components */
   for(int i=0; i < proactiveComponents.size; i++) {
     Component *c = proactiveComponents.elements[i];
     int n = c->npro(), *p = c->pro();
     double (Component::**g)() = c->g();
     Transition *t = c->t();
     double d;

     /* ... for all proactions in current component starting in
state q */
     for(int j=0; j<n; j++) {
       if (c->q != t[p[j]].from)
      continue;
      d = (c->*g[p[j]])();
      // ... check if there is an enabled proaction that can
happen before
        // firingDelay
      if (0 <= d && (firingDelay < 0 || d < firingDelay)) {
      scheduledComponent = c;
      scheduledTransition = p[j];
      firingDelay = d;
      }
      if (firingDelay == 0)
     break;
    }
    if (firingDelay == 0)
      break;
  }
}
```

The DEM::waitForInterrupt Functions

```
int
DEM::waitForInterrupt()
{
  int selection;

#ifndef WIN
```

```
    struct timespec timeout;
  #endif

  #ifndef EMBEDDED
    char error[256];
  #endif

    fd_set rfd;
    FD_ZERO(&rfd);

    if (interruptAlerts.size > 0)
      // If the alerts from interrupt list is not empty
      selection = 0;
    else if (firingDelay == 0)
      // If firingDelay is zero then scheduledTransition must be
taken
      // immediately.
      selection = 0;
    else if (firingDelay > 0) {

#ifndef EMBEDDED
      // if firingDelay is > 0 then scheduledTransition is
scheduled in
      // the future: what to do depends on RT and SIM options.
      if (realtimeEnabled == 1) {
        if (simulationEnabled == 0) {
#endif // EMBEDDED

          // If firingDelay > 0, rt on, sim off then wait with
timeout
          // at (firingDelay - computationTime).
          double computationTime = dem_time() -
currentApplicationTime;
          double timeToWait =
            (firingDelay-computationTime)<0 ? 0 : (firingDelay-
computationTime);

#ifdef WIN
          Sleep((DWORD) (timeToWait*1000));
          // Sleep doesn't return any values, so there's no way to
tell if an
          // interrupt was received at this point. Since this
function is used
          // only on embedded version of Teja for Windows we assume
that no
          // interrupts were received.
          selection = 0;
#else
          timeout.tv_sec = (long) timeToWait;
          timeout.tv_nsec =
            (long) ((timeToWait - timeout.tv_sec)*1000000000);
          selection = (nanosleep(&timeout, 0) == -1) ? 1 : 0;
  #endif // WIN

  #ifndef EMBEDDED
        }
        else {
       // If firingDelay > 0, rt on, sim on then
 scheduledTransition
       // is taken immediately.
       //
       selection = 0;
       }
     }
     else {
       // If firingDelay >0, rt off, (sim off or on) then no
 realtime
       // license is available on the system.
 scheduledTransition
       // should happen in the future, but, since there's no rt
       // license, it's just ignored (an error message is
 printed).

       writeErrorLog("Realtime event scheduling option not in
 license.\n");
       sprintf(error,
            "Proactive transition %d in %s %ld after %f seconds
 ignored.\n",
            scheduledTransition,
            ClassDescription.elements[scheduledComponent-
>cid()]
            ->className,
            scheduledComponent->id,
            firingDelay);
      writeErrorLog(error);
      (void) select(FD_SETSIZE, &rfd, 0, 0, 0);
      selection = 1;
    }
#endif // EMBEDDED

  }
  else {
    // If firingDelay < 0 then no proaction is enabled. Select
without
    // timeout.
    (void) select(FD_SETSIZE, &rfd, 0, 0, 0);
    selection = 1;
  }

  return selection;
}

int
DEM::waitForInterrupt(fd_set * rfd)
```

```
 {
   int selection = 0;
   struct timeval timeout;

 #ifndef EMBEDDED
   char error[256];

   setRfd(rfd);
 #else
   FD_ZERO(rfd);
 #endif // EMBEDDED

   if (interruptAlerts.size > 0)
     // If the alerts from interrupt list is not empty
     selection = 0;
   else if (firingDelay == 0)
     // If the firingDelay is zero then scheduledTransition
must be
     // taken immediately.
     selection = 0;
   else if (firingDelay > 0) {

#ifndef EMBEDDED
     // if firingDelay is > 0 then scheduledTransition is
scheduled in
     // the future: what to do depends on RT and SIM options.
     if (realtimeEnabled == 1) {
       if (simulationEnabled == 0) {
#endif // EMBEDDED

       // If firingDelay > 0, rt on, sim off then select with
timeout
       // at (firingDelay - computationTime).
       double computationTime = dem_time() -
currentApplicationTime;
       double timeToWait =
         (firingDelay-computationTime)<0 ? 0 : (firingDelay-
computationTime);

       timeout.tv_sec = (long) timeToWait;
       timeout.tv_usec =
         (long) ((timeToWait - timeout.tv_sec)*1000000);
       selection = select(FD_SETSIZE, rfd, 0, 0, &timeout);

#ifndef EMBEDDED
       }
       else {
       // If firingDelay > 0, rt on, sim on then
scheduledTransition
       // is taken immediately.
       selection = 0;
       }
```

```
    }
    else {
      // If firingDelay >0, rt off, (sim off or on) then no
realtime
      // license is available on the system.
scheduledTransition
      // should happen in the future, but, since there's no rt
      // license, it's just ignored (an error message is
printed).
      // Select without timeout.

      writeErrorLog("Realtime event scheduling option not in
license.\n");
      sprintf(error,
           "Proactive transition %d in %s %ld after %f seconds
ignored.\n",
           scheduledTransition,
           ClassDescription.elements[scheduledComponent-
>cid()]
           ->className,
           scheduledComponent->id,
           firingDelay);
      writeErrorLog(error);

      selection = select(FD_SETSIZE, rfd, 0, 0, 0);
    }
#endif // EMBEDDED

  }
  else {
    // If firingDelay < 0 then no proaction is enabled. Select
without
    // timeout.

    selection = select(FD_SETSIZE, rfd, 0, 0, 0);
  }

  return selection;

}
```

The DEM::updateTimeVariables Function

```
void
DEM::updateTimeVariables (int selection)
{

#ifndef EMBEDDED
  if (simulationEnabled == 0) {
#endif // EMBEDDED
    if (selection == 0) {
      // Sim is off and timeout expired
      transitionRealTime = dem_time();
      slip = transitionRealTime - (currentApplicationTime +
firingDelay);
      firingDelay = transitionRealTime -
currentApplicationTime;
      currentApplicationTime = transitionRealTime;
    }
    else {
      // Sim is off and alert or interrupt was received
      transitionRealTime = dem_time();
      slip = 0;
      firingDelay = transitionRealTime -
currentApplicationTime;
      currentApplicationTime = transitionRealTime;
    }
#ifndef EMBEDDED
  }
  else {
    if ((selection == 0) && (interruptAlerts.size == 0)) {
      // Sim is on, selection is 0 and no interrupt was
received
      transitionRealTime = currentApplicationTime +
firingDelay;
      slip = 0;
      currentApplicationTime = transitionRealTime;
    }
    else {
      // Sim is on and selection is <> 0 (an alert or an
interrupt has
      // been received) was received
      transitionRealTime = currentApplicationTime;
      slip = 0;
      firingDelay = 0;
    }
  }
#endif // EMBEDDED
}
```

The Component::update Function

```
void
Component::update(double elapsedTime)
{
  for(int i=0; i<this->ncs(); i++)
    x[i] += elapsedTime*xdot[i];
}
```